Lab 4: Port Scanner

Due December 4th at 11:00pm

Project goals

Port scanners like Nmap[1] can probe remote hosts for open ports and are usually used by network administrators to verify the security of machines in their network. Port scanners are also popular amongst Internet miscreants who use them to find hosts to compromise by probing for software versions with known vulnerabilities.

In this project you will create a basic port scanner with IPv4 support that is written for network administrators interested in ensuring that machines on their network run only expected services. In addition to helping you gain experience with socket programming, this project will help you appreciate the interplay of various implementation of firewalls, transport protocols, and operating systems.

Deliverables

Your submission should include the following files:

- `portScanner.c or portScanner.cpp`
- `Supporting files (*.c, *.cpp, *.h)`
- `Makefile`
- `README`
- `Write-up in a PDF format`

Your `README` file should contain a short header containing your name, username, and the assignment title. The `README` should additionally contain a short description of your code, tasks accomplished, and how to compile, execute, and interpret the output of your program.

Even though you are working in groups on this assignment, each student must submit their own `write-up`. You may not see your partner's written answers. The `write-up` PDF file should contain a short header containing your name, username, and the assignment title. It should also contain general information about the contents of each file that you submit. That is, a person not familiar with this assignment should have a clear understanding of the purpose of each submitted file after reading your `write-up`. Furthermore, the `write-up` should include a 'Credits:' section, which lists all the people (e.g., students in the class or not, friends, colleagues, etc.) and resources (e.g., web pages, manuals, books, code, etc.) that you consulted for this assignment. Finally, the `write-up` should contain team spirit scores for your partner(s).

[1] `http://nmap.org`

Milestone

The entire assignment is due on December 4th, but your group will also submit a Milestone within one week of the project start to show you have read the assignment and are on track to finish the project. More details about what is required for the Milestone can be found later in this document.

Important notes

- You can use either C or C++ (an object-oriented programming language might be a better choice for this particular project).
- Your program should always exit gracefully – points will be taken off for program crashes, segmentation faults, and memory leaks.
- The code should be clean and well-commented.
- There should be no magic numbers in your code.
- Points will be taken off for failing to protect function calls which are not thread-safe.
- Do not protect known thread-safe functions.
- Break the code into functions if you find yourself using more than 5 levels of indentation.
- The submitted code must compile and run on the `Blondie` VM.
- If the code does not run on `Blondie` during the scheduled demo, the group will be allowed to demo the project on the machine of their choice. This carries a 15-point penalty.
- If team members are unable to demo their project during the scheduled demo slot (i.e. the code does not run on `Blondie` and none of the team members have a computer in their possession which is capable of running the code), they will receive a failing grade for the project. No exceptions. The actual grade will depend on the submitted code and the ability to answer project-related questions.
- If team members are unable to demo their project during the scheduled demo slot (i.e. the code does not run on `Blondie` and none of the team members have a computer in their possession which is capable of running the code), they will receive a failing grade for the project. No exceptions. The actual grade will depend on the submitted code and the ability to answer project-related questions.

  Yes – we wrote this twice.
- Due to the nature of this project, you will not be allowed to make any changes to the submitted code before or during the demo. What you submit on `OnCourse` is what we will grade. No exceptions.
- You must schedule an appointment to demonstrate your project. Groups that fail to demonstrate their project will not receive any credit.
- You should be able to answer any questions about any part of the code during the demonstration.

Cheating and Plagiarism

Although it is okay to discuss assignments (verbally; no written exchanges) with other students, all solutions handed in must be your own. Your code must be a group effort between the two group members only (you are expected to write the code together). The `write-up` must be written individually; you may **not** read your project partner's (or any other student's) `write-up`.

Copying answers from friends or the Internet is prohibited. We will use different software packages to check for plagiarism. Furthermore, you must cite all sources used for problem solutions (i.e. websites, books, discussions with classmates, etc.) in the `write-up`. For more information please see Indiana University Code of Student Rights, Responsibilities, and Conduct: `http://www.indiana.edu/~code/`.

Submission Instructions

Submit a tarball with your code in a folder named *Project 4* via `OnCourse`. Submit `write-up` along with your code, but do not put it in the archive.

Submission example:

```
project4.tar
write-up.pdf
```

1 Scanner Details

There are two important aspects to any scanning program: 1) the port states it can detect and 2) the techniques it uses to make its inference.

1.1 Port States

We expect your program to recognize the following five port states:

1. **Open:** This state implies that the application is actively accepting TCP connections or UDP datagrams on this port. Finding these is the primary goal of your program.

2. **Closed:** This state implies that the port is accessible to the scanner but there is no application listening on it.

3. **Filtered:** Filtering devices, such as firewalls, can prevent scanners from inferring if a port is open. In such cases, your program should try to infer if a port is filtered.

4. **Unfiltered:** This state means that the port is accessible but no inference can be made on whether it is open or closed.

5. **Open | Filtered:** There are cases when open ports give no response. In such cases, the lack of response can either imply that the port is open or that it is filtered.

1.2 Port Scanning Techniques

Most of the scan types mentioned below are only available to privileged users since scanners send and receive raw packets, which requires root access on Unix systems. Also, note that each scan type is limited in what it can infer.

1. **TCP SYN scan (Half-Open scan):** This is the most popular scanning option since it is fast and unrestricted by firewalls that want to permit connecting on certain ports only. It is also relatively stealthy since it never completes TCP connections. The basic idea of this technique is simple: you send a TCP SYN packet, as if you were going to open a real connection, and then wait for a response. A SYN/ACK indicates that the port is open while a RST indicates that the port is closed. If no response is received after several retransmissions, the port should be marked as filtered. The port is also marked as filtered if an ICMP unreachable error (type 3, code 1, 2, 3, 9, 10, or 13) is received. Finally, since a rare feature of TCP can sometimes just send the SYN packet in response to a SYN, mark the port as open in that case as well.

2. **TCP NULL, FIN, and Xmas scans:** The TCP specification says that if a port is closed, it should send back an RST in response to a packet containing anything other than a SYN, ACK, or RST. No response is expected when the port is open. For machines compliant with this specification, scanners can do NULL scan (no bits of TCP flags are set), FIN scan (just the FIN bit is set) or Xmas scan (sets the FIN, PSH and URG flags, lighting the packet like a Christmas tree) to infer the state of the port. Note, however, that since firewall devices can block responses, lack of response can only mean that the port is open | filtered. Further, in cases where a firewall device sends an ICMP unreachable error (type 3, code 1, 2, 3, 9, 10 or 13), the port can be marked filtered.

3. **TCP ACK scan:** An ACK scan probe packet only has the ACK flag set. In the lack of firewall devices, both open and closed ports will return an RST packet and a scanner can conclude that the port is unfiltered. Ports that do not respond or send ICMP error messages back (type 3, code 1, 2, 3, 9, 10 or 13) are labeled filtered. Note that this scan cannot infer whether a port is open or closed.

4. **UDP scan:** Scanning UDP ports is more difficult than scanning TCP ports since UDP is a connectionless protocol. One cannot be certain whether a UDP port is listening or not because no response is required by the specification of UDP. So, proceed as follows: for DNS (port 53), send a protocol-specific payload, but for other ports, leave the packet empty or send a fixed-length random payload (experiment with both). If an ICMP port unreachable error (type 3, code 3) is returned, mark the port as closed. For other ICMP unreachable errors (type 3, codes 1, 2, 9, 10, or 13) mark the port as filtered. Occasionally, a service will respond with a UDP packet, proving that it is open. If no response is received after retransmissions, the port is classified as open | filtered.

2 Project Specification

The basic idea behind a port scanner is simple: given an IP address of a machine and a list of interesting TCP or UDP ports to scan, the scanner will connect on each port using TCP or UDP sockets, make a determination of whether or not the port is open based on success of the connection request and close the socket before moving on to the next port to scan.

2.1 Specifics of Expected Port Scanner Functionality

Your port scanner should run on the Blondie VM and must be written in C/C++. An administrator would invoke it as: "./portScanner [option1, ..., optionN]". Implement the following options:

- −−help. Example: "./portScanner −−help".
- −−ports <ports to scan>. Example: "./portScanner −−ports 1,2,3-5".
- −−ip <IP address to scan>. Example: "./portScanner −−ip 127.0.0.1".
- −−prefix <IP prefix to scan>. Example: "./portScanner −−prefix 127.143.151.123/24".
- −−file <file name containing IP addresses to scan>. Example: "./portScanner −−file filename.txt".
- −−speedup <parallel threads to use>. Example: "./portScanner −−speedup 10".
- −−scan <one or more scans>. Example: "./portScanner −−scan SYN NULL FIN XMAS".

Details of each option are given below:

- **help:** When portScanner is invoked with this option, it should display the various options available to the user.
- **ports:** Your portScanner will scan ports [1-1024] by default. However, if this option is specified, it will scan ports specified on the command line. The latter could be individual ports separated by a comma or a range.

- **ip/prefix/file:** These options allow a user to scan an individual IP address, an IP prefix, or a list of IP addresses from a file respectively. When IP addresses are specified in a file, you can assume them to be one on each line. A user may invoke the portScanner with more than one of these options. If none of these options are specified, check for the presence of an individual IP address as an argument. If that check also fails, flag an error and ask the user to try again.
- **speedup:** The code should be single threaded by default. This would keep the implementation simple, but it will be slow. Specifying this option will allow a user to use the faster, multi-threaded version of the portScanner. The user will specify the number of threads to be used.
- **scan:** The portScanner will perform all scans by default. However, the user may select any subset of scans that they wish. This is done via the following flags: SYN, NULL, FIN, XMAS, ACK, UDP. Multiple of these may be specified implying that each of the listed scans should be run.

For example, the following command line options tell the scanner to use 5 threads to scan the first 1,000 ports of 74.125.225.68 via UDP and SYN scans.

```
# ./portScanner --ports 1,2,3-1000 --ip 74.125.225.68 --speedup 5 --scan UDP SYN
```

Your portScanner must support the options specified above exactly as they are written.

Below are some examples of command-line options' formats that **are not accepted**:

```
# ./portScanner --ports 1, 2, 3 - 1000 --ip 74.125.225.68 --speedup 5 --scan UDP SYN
# ./portScanner --ports 1,2,3-1000 --ip "74.125.225.68" --speedup 5 --scan UDP SYN
# ./portScanner --ports 1,2,3-1000 --ip 74.125.225.68 --speedup 5 --scan "UDP SYN"
# ./portScanner --ports 1,2,3-1000 --ip 74.125.225.68 --speedup 5 --scan UDP, SYN
```

2.2 Verifying standard services

For SSH, HTTP, SMTP[2], POP, IMAP, and WHOIS, your program should verify that the port is indeed running the standard service expected on that port. Upon verification, find out the specific version of the software as well. Inferring this information will require you to do one of the following: 1) parse identifying information sent by the service if it sends it or 2) send appropriate queries to cause the service to reveal this information to you. For example, SSH sends you the version information immediately upon connection but for HTTP you must send a valid query to the remote server and read back the header of its response to find the version of the software it is running. You are allowed to use the `connect()` system call for this part of the assignment.

Some hosts will not send you a version number of the running software. Consequently, we only require you to demonstrate that your port scanner is able to correctly detect the software version on any host of your choice. Make sure to bring the list of hosts to the demo.

[2]You can use either one of the following ports for SMTP: 24, 25, 587.

2.3 Timeouts and malicious targets

As the machines you are expected to scan are not under your control, your program should be designed to handle arbitrarily bad responses from the remote machine. No response from the remote machine should cause your program to crash. Likewise, your program should timeout in a reasonable period of time (generally a handful of seconds) no matter how the remote machine responds.

2.4 Expectations on concurrency

We require in the speedup option that multiple threads be spawned to divide the work load. Threads should not be idle due to the static division of large amounts of work. For example, if requested to use 5 threads to scan 5,000 ports, you should not simply initially assign each thread to scan 1,000 ports - some threads may finish much faster, and should then assist the slower running threads. Additionally, your program should not continually create new threads. For example, you should not create a thread to scan a port, scan the port, record the results, and then destroy the thread only to repeat the process for the next port. No possible execution paths of these concurrent threads should lead to an error. Specifically, ensure that you protect all functions and library calls explicitly unless they are noted to be thread safe. All memory should be freed when your port scanner program exits. Specifically, there should not be any memory leaks or zombie threads. Do not rely on the OS to clean up terminated threads.

2.5 Output

After each invocation, the portScanner should output a succinct summary of the scanned ports for each IP address. Additionally, for each port in range [1-1024], the summary should include the name of the service associated with that port. To find services associated with ports [1-1024], visit `http://www.iana.org/assignments/port-numbers`.

When multiple scans are used, the results for each scan should be included in the output. Additionally, the combined inference from all of the scans that were run should be listed. Doing this requires you to determine a way to combine possibly contradictory results from different scans. Be prepared to justify your method to the evaluator during the code review.

The output produced by your program should be easy to read and understand. Sample output:

```
# ./portScanner --file ip_list.txt --ports 15,7,9,89,80 --scan UDP SYN
```

```
Scanning...

Scan took 177.949 seconds.

IP address: 129.79.78.193
Open ports:
Port      Service Name (if applicable)          Results                                 Conclusion
-------------------------------------------------------------------------------------------------
80        World Wide Web HTTP                   UDP(Open|Filtered)    SYN(Open)           OPEN

Closed/Filtered/Unfiltered ports:
Port      Service Name (if applicable)          Results                                 Conclusion
-------------------------------------------------------------------------------------------------
7         Echo                                  UDP(Open|Filtered)    SYN(Filtered)       Filtered
9         Discard                               UDP(Open|Filtered)    SYN(Filtered)       Filtered
15        Unassigned                            UDP(Open|Filtered)    SYN(Filtered)       Filtered
89        SU/MIT Telnet Gateway                 UDP(Open|Filtered)    SYN(Filtered)       Filtered

IP address: 128.210.7.199
Open ports:
Port      Service Name (if applicable)          Results                                 Conclusion
-------------------------------------------------------------------------------------------------
80        World Wide Web HTTP                   UDP(Open|Filtered)    SYN(Open)           OPEN

Closed/Filtered/Unfiltered ports:
Port      Service Name (if applicable)          Results                                 Conclusion
-------------------------------------------------------------------------------------------------
7         Echo                                  UDP(Open|Filtered)    SYN(Closed)         Closed
9         Discard                               UDP(Open|Filtered)    SYN(Closed)         Closed
15        Unassigned                            UDP(Open|Filtered)    SYN(Closed)         Closed
89        SU/MIT Telnet Gateway                 UDP(Open|Filtered)    SYN(Closed)         Closed
```

3 Resources and Restrictions

3.1 Getting Started

Begin by familiarizing yourself with the Nmap software. A simple starting point is to scan your machine, aka, localhost, via "nmap 127.0.0.1". Another useful resource is telnet, which will allow you to interact with a server using a plain text command line. For example, "telnet burrow.cs.indiana.edu 22" will allow you to connect to the SSH service running on burrow.cs.indiana.edu on port 22. burrow.cs.indiana.edu will respond by telling you a bit about the SSH service and will wait for you to send the appropriate authentication.

3.2 Implementation Specifics

As stated earlier, you are required to use either C or C++ for this project. Additionally, you must use the native Linux/BSD socket system calls to open sockets. You may not use other socket libraries. No credit will be given to solutions that do not adhere to these requirements. These restrictions are being made so you become familiar with the details of lower-level socket programming. If you choose to do so, you may use `libpcap` to capture packets, although you should still create/analyze the packets manually.

To create a multi-threaded version of your program, use the `pthreads` library. This library can be enabled

with the gcc compiler via -pthread option. Additionally, the -D_REENTRANT gcc option, which makes code libraries re-entrant, may come in handy because it can help the compiler load copies of libraries where it is safe to have two threads call the same function.

The following system calls will likely be required to complete the assignment: **connect**, **htons**, **pthread_create**, **pthread_join**, **pthread_kill**, **recv**, **send**, **setsockopt**, **sleep**, **socket**. You may also want to look at the more advanced synchronization methods available in pthreads.

You `cannot` use any socket/threading/capture/input parsing libraries that were not specified in this document. That is, you are allowed to use `pcap`, `pthreads`, and `raw sockets`, but cannot use `boost`, `TinyThread`, etc. In addition, although you may use `pcap` to receive packets, you may not use it to create or send packets. Finally, you cannot copy the checksum code from the Internet – we expect you to know the algorithm and be able to implement it yourself.

You will probably find protocol headers from `netinet/*` useful for this assignment. In addition, you will most likely have to copy DNS headers from the Internet. This is the `only` part of this assignment that you are allowed to copy.

3.3 Experiment with Caution

Exercise frugality in testing your program since system administrators of organizations whose machines you scan would likely get upset and complain to our system administrators, who in turn may have to limit the scope of the project beyond the point where you would find it a useful practical experience. In addition, `do not` try to scan any non-SOIC machines while you are on campus.

3.4 Roadmap

Build your program incrementally. Here is a road-map to use:

1. Get your program to scan one TCP port on an IP address using one scanning technique and display the entire output on the screen.
2. Expand the functionality for other TCP port scanning techniques.
3. Add scanning of UDP sockets.
4. Parse the output to derive conclusions about which ports are open.
5. Expand the program to scan ports in a loop.
6. Verify that ports for SSH, HTTP, SMTP, POP, IMAP, and WHOIS are indeed running these services.
7. Add functionality for options related to IP addresses and ports to scan.
8. Develop a multi-threaded version of the program.
9. Detect and recover from failed/hanging threads.

3.5 Suggestions/clarifications

- Start coding early.
- You will find `http://www.ipaddressguide.com/cidr` useful for this assignment.
- Become familiar with `nmap`, `Wireshark`, and `tcpdump`. You will find these tools very useful in the course of this project.
- Do not mindlessly copy code from your friends or the Internet. First of all, doing so is prohibited and violates Indiana University's code of student conduct. Second, failing to answer any questions about the code during the demo will result in getting 0 credit for that portion of the assignment.
- Your port scanner should be relatively fast. If it takes a single-threaded version of your program more than 5 minutes to scan 10 ports of one IP address, something is not right.
- If the host does not respond after 4-7 seconds: re-send the packet (up to 3 times) and wait 4-7 seconds for the response after each attempt. If there is still no response, make the decision about the port status and move on to the next port.
- Do not hardcode the address of the main network card - your port scanner should determine it at run time.
- You must use raw sockets to send TCP packets.
- Although you may use a datagram socket to send DNS packets (port 53), you must use raw sockets to send packets to all other UDP ports.
- You may not use any external data structure libraries - you will need to write your own in case you decide to stick with C.
- You must use either raw sockets or libpcap to receive incoming packets. In case you choose to go the easier route and use libpcap, make sure that you do not spawn multiple libpcap instances - one is enough. For example, if your port scanner is using 100 scanning threads to scan a range of IP addresses, there should be exactly one libpcap listening thread (and not 100).
- You should consider both a network address (also called network ID) and a broadcast address when scanning an IP prefix. For example, if your port scanner is supplied with the 192.168.5.12/30 prefix, it should scan the following IP addresses:
  192.168.5.12
  192.168.5.13
  192.168.5.14
  192.168.5.15

4 Testing

4.1 Blondie

SOIC has allocated the Blondie VM (`blondie.soic.indiana.edu`) that you can use to run your code. All students currently enrolled in P538 have `sudo` rights on this VM. Do not abuse this privilege! For

example, you should not try to scan 100,000 IP addresses with 1,000 threads. Doing so could exhaust the available resources and all students may lose access to the VM until it is rebooted (which could take some time).

When using Blondie, you should always `nice` your port scanners like so:

```
$ sudo nice ./portScanner
```

4.2 Dagwood

SOIC has also allocated the Dagwood VM (`dagwood.soic.indiana.edu`) that you are encouraged to use to test your code. The purpose of the Dagwood VM is to provide you with an on-campus machine which should give the same responses regardless of how many times you scan it (which is not the case with some of the hosts on the Internet).

While we encourage you to use Dagwood as much as possible, you should also test your port scanners on other hosts.

Table 1 shows services that are currently running on Dagwood.

Service Name	Port
The Secure Shell (SSH) Protocol	22
any private mail system	24
WHOIS	43
HTTP	80
Post Office Protocol - Version 3	110
Internet Message Access Protocol	143

Table 1: Dagwood services

5 Deliverables and grading

- **Milestone 1**

  For Milestone 1, we expect you to be able to parse all command line arguments, read IP addresses from input files, and convert the supplied IP prefixes to a list of individual IP addresses. In order for us to easily test this milestone, we ask that you print out to the terminal the list of IP addresses that your code was supplied with.

  For example, let's assume that the input file, *ipAddresses.txt*, contains the following lines:

  ```
  127.0.0.17
  127.0.0.18
  ```

  When invoked with the command

  ```
  $ ./portScanner --file ipAddresses.txt --prefix 127.0.0.1/30
  ```

  your code should print out:

```
127.0.0.0
127.0.0.1
127.0.0.2
127.0.0.3
127.0.0.17
127.0.0.18
```

Once you have done the above, you should put into place a roadmap of what else you must do to finish this project. Break your goals into chunks – TCP scans, multithreading, etc. – and submit the roadmap PDF on `OnCourse` along with your code. Please note, your Milestone 1 submission must include a working Makefile.

Only one person in the group needs to submit a roadmap and base code. However, failure to submit Milestone 1 will result in a 10% deduction on the final grade for this project. Please note that submissions do not need to be perfect – we are simply looking to see that you have started this project and have a concrete plan of what needs to be done in the next weeks.

- **Code submission and demo**

  Submit: 1) your `code` as a single archive file (.tar or .tar.gz file formats only), and 2) `write-up` PDF via OnCourse by 11:00pm of the day of the deadline. Shortly after the submission deadline, demo slots will be posted on the Demonstration Scheduling System. You must schedule an appointment to demonstrate your project. Groups that fail to demonstrate their project will not receive credit for the project. If a group member fails to attend his or her scheduled demonstration time slot, it will result in a 10 point reduction in his or her grade.

- **Grade distribution (subject to change)**

Deliverable	**Grade %**
The port scanner can scan any port on specified IP address via all TCP scanning techniques.	20
The port scanner can perform UDP scan of any port on specified IP address.	20
The code can analyze the incoming packets and derive conclusions about which ports are open/closed/filtered/unfiltered.	15
The code can verify that ports for SSH, HTTP, SMTP, POP, IMAP, and WHOIS are indeed running these services and retrieve the actual service versions.	12
The port scanner can scan IP prefixes and read IP addresses from files.	11
The code is multi-threaded.	22

Table 2: Grade distribution

6 Credits

This lab is © Minaxi Gupta, which we have modified for P-538.